Practical Shadow Mapping

Stefan Brabec Thomas Annen Hans-Peter Seidel

Max-Planck-Institut für Informatik
Saarbrücken, Germany

Abstract

In this paper we propose several methods that can greatly improve image quality when using the shadow mapping algorithm. Shadow artifacts introduced by shadow mapping are mainly due to low resolution shadow maps and/or the limited numerical precision used when performing the shadow test. These problems especially arise when the light source's viewing frustum, from which the shadow map is generated, is not adjusted to the actual camera view. We show how a tight fitting frustum can be computed such that the shadow mapping algorithm concentrates on the visible parts of the scene and takes advantage of nearly the full available precision. Furthermore, we recommend uniformly spaced depth values in contrast to perspectively spaced depths in order to equally sample the scene seen from the light source.

1. Introduction

Shadow mapping [10, 7] is one of the most common shadow techniques used for real time rendering. In recent years, dedicated hardware support for shadow mapping (high precision depth textures, shadow test functionality) made its way from high-end visualization systems to consumer class graphics hardware. In order to obtain visually pleasing shadows one has to be careful about sampling artifacts that are likely to occur. These artifacts especially arise when the light source's viewing frustum is not adapted to the current camera view. Recently, two papers addressed this issue. Fernando et al. [2] came up with a method called *Adaptive Shadow Maps* (ASMs) where they proposed a hierarchical refinement structure that adaptively generates shadow maps based on the camera view. ASMs can be used for hardware-accelerated rendering but require many rendering passes in order to refine the shadow map. Stamminger et al. [8] showed that it is also possible to compute shadow maps in the post-perspective space (normalized

device coordinates) of the current camera view. These *Perspective Shadow Maps* (PSMs) can be directly implemented in hardware and greatly reduce shadow map aliasing.

In this paper we focus on the traditional shadow map algorithm and show how the light source's viewing frustum can be adjusted to use most of the available precision, in terms of shadow map resolution. Since it is also important that the available depth precision is used equally for all regions inside this frustum, we show how uniformly spaced depth values can be used when generating the shadow map.

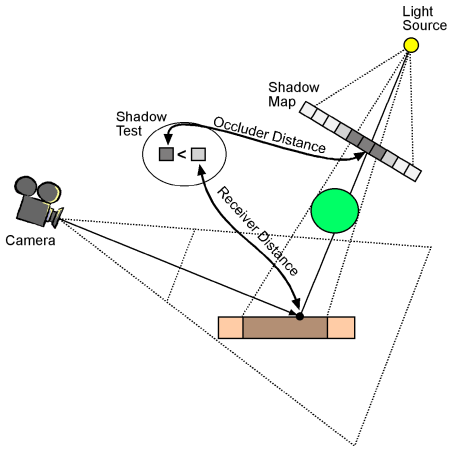

Figure 1. Shadow mapping illustrated.

Let us first recall how shadow mapping can be implemented using graphics hardware. In a first step, the scene is rendered as seen by the light source. Using the z-buffer we obtain the depth values of the frontmost pixels which are then stored away in the so called *shadow map*. In the second step the scene is rendered once again, this time from the camera's point of view. To check whether a given pixel is in shadow (with respect to the light source) we transform the pixel's coordinates to the light source's coordinate system. By comparing the resulting distance value with the corresponding value stored in the shadow map we can check if a pixel is in shadow (stored depth is less than actual depth) or lit by the light source (stored depth is greater or equal). This comparison step is illustrated in Figure 1. Shadow mapping can be efficiently and directly implemented on graphics hardware [7] with

- an internal texture format to store shadow maps (usually the precision of this format should match the precision of the z-buffer), and
- a comparison mechanism to perform the shadow test.

Using OpenGL as the underlying graphics API, this functionality is implemented by two extensions. One is the `depth_texture` extension which allows storing the z-buffer's depth values as a texture image (usually 16 or 24 bit precision). The actual comparison step (`shadow` extension) is done via homogeneous texture coordinates. OpenGL provides a special mode which automatically computes texture coordinates which correspond to vertex eye coordinates (camera coordinates). These coordinates can then be transformed to the light source texture space. After that, the shadow extension compares the shadow map entry at $(s/q, t/q)$ with the transformed z value r/q and returns a color of black or white, depending on the result of the comparison.

2. Distribution of depth values

When rendering the scene from a given viewpoint, depth values are sampled non-uniformly ($1/z$) due to the perspective projection. This makes sense for the camera position, since objects near to the viewer are more important than those far away, and therefore sampled at a higher precision. For the light source position this assumption is no longer true. It could be the case that objects very far from the light source are the main focus of the actual camera, so sampling those at lower z precision may introduce artifacts, e.g. missing shadow detail. A solution to this was e.g. proposed by Heidrich [3]. Here depth values are sampled uniformly using a 1D ramp texture that maps eye space depth values to color values, which are later used as the corresponding shadow map.

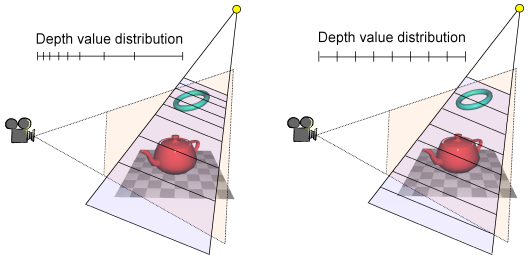

Figure 2. Distribution of depth values. Left: $1/z$ Right: z linear.

Figure 2 illustrates the difference between linear and $1/z$ mapping. On the left side, depth values are sampled using the traditional perspective projection. Objects near to the light source obtain most of the available depth values, whereas objects far away (e.g. the ground plane) have less precision. Shadow details for the teapot may be missing while the torus may be oversampled. The right side of Figure 2 shows the same setup using a linear distribution of depth values. Here all objects are sampled equally. We can achieve this linear distribution of depth values using a customized vertex transformation, which can be implemented using the so called *vertex shader* or *vertex program* functionality [4] available on all recent graphics cards.

Instead of transforming all components of a homogeneous point $P = (x_e, y_e, z_e, w_e)$ by the perspective transformation matrix, e.g. $(x, y, z, w) = Light_{proj} \cdot P$, we replace the z component by a new value $z' = z_l * w$. The linear depth value $z_l \in [0; 1]$ corresponds to the eye space value z_e mapped according to the light source near and far plane:

$$z_l = -\frac{z_e + near}{far - near}$$

To account for normalization $(x/w, y/w, z/w, 1)$ which takes place afterwards (normalized device coordinates), we also pre-multiply z_l by w. This way, the z component is not affected by the perspective division, and depth values are uniformly distributed between the near and far plane.

3. How near, how far ?

Another very important property that affects the depth precision of the shadow map is the setting of the near and far plane when rendering from the light source position. A common approach is to set those to nearly arbitrary values like 0.01 and 1000.0 and hope that a shadow map with 24 bit precision will still cover enough of the relevant shadow information.

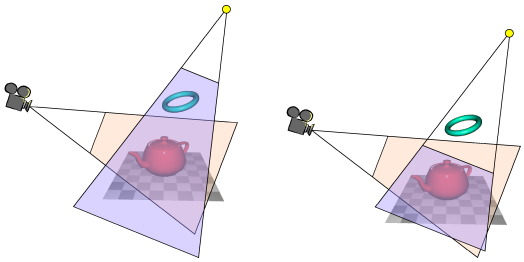

Figure 3. Left: Standard near/far setup. Right: Tight fitting near/far setup.

By analyzing the scene we can improve depth precision by setting the near and far plane such that all relevant objects are inside the light's viewing frustum, as depicted on the left side of Figure 3.

In terms of depth precision, this setting is still far from being optimal. It is clear that the torus needs to be included in the shadow map, since it will cast a large shadow onto the ground plane and teapot, but for this shadow information only one bit would be sufficient since the shadow caster itself is not seen by the camera. So what we really would like to have is some kind of tight fitting near and far plane setup that concentrates on those objects that are visible in the final scene (seen from camera position). This optimal setting is depicted on the right side of Figure 3. If we would render this scene with the traditional approach, shadows cast by the torus would be missing since the whole object lies outside the light's viewing frustum and would be clipped away.

We can easily include such objects by having depth values of objects in front of the near or beyond the far plane clamped to zero or one, respectively. This clamping can be achieved with a special *depth replace* texture mode, available as part of the *texture shader* extension provided by recent NVIDIA graphics cards [5][1].

Assume we want to render a shadow map with 16 bit precision where depth values outside the valid range are clamped rather than clipped away. These depth values can be encoded using two bytes, where one contains the least significant bits (LSB) while the other stores the most significant bits (MSB). If we setup a two dimensional ramp texture in which we encode the LSBs in the red channel (0 to 255, column) and in the green channel we store the MSBs (0 to 255, row position), we can map the lower 8 bit of a given z value by setting the s coordinate to $256.0 * z$ and using the s coordinate repeat mode. This way s maps the fractional part of $256.0 * z$ to the LSB entry in the

[1] As the name implies, this texture mode replaces a fragment's window space depth value.

ramp texture. To code the MSB we can directly map z to t and use a *clamp-to-edge* mode such that values $z < 0$ are clamped to 0 and values $z > 1.0$ are clamped to 1.0. To replace the current window space depth value with the new, clamped value we now have to setup the texture shader as depicted in Figure 4. Texture unit 0 is responsible for mapping the higher and lower bits of the depth value to a color encoded RGB value (blue component set to zero). Texture unit 1 is configured to perform a *dot product depth replace* operation, which takes texture coordinates from unit 1 and computes a dot product with the result of the previous texture unit (color encoded depth). The result is a new depth value that is just a clamped version of the original depth value.

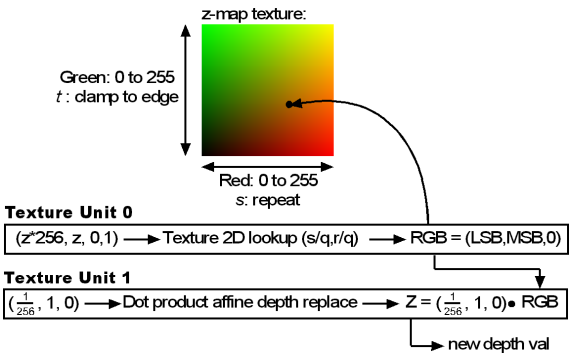

Figure 4. Texture shader setup.

One problem with this texture shader setup is that the LSB is repeated even for objects in front or beyond the near/far planes, due to the s coordinate texture repeat. If we set the planes such that all pixels in front of the near clipping plane are mapped to a MSB of 0 and pixels beyond the far clipping plane to a MSB of 255 we do not have to worry about the LSB part of the depth value. So the effective range of depth values is between 0x0100 and 0xfeff.

Up to now we did not take care about the view frustum culling that takes place before the rasterization. If for example an object lies completely in front of the near clipping plane all triangles would be culled away after the transformation step (clip coordinates). To avoid this we simply modify the vertex shader described in Section 2 such that the z component of the output position is set to a value of $0.5 * w$. This way all vertices are forced to lie between the valid $[0; 1]$ z range. The z values passed as texture coordinates for texture unit 0 are still the linear z_l's. After the depth replace step we then restore valid z coordinates used for depth testing.

This method can be extended to 24 bit depths by using a special HILO texture format [5], for which filtering takes place at 16 bits per component. Here we encode the lower 16 bits of z as the LO-component (0x0000-0xffff) and the remaining upper 8 bits as the HI-component (0x00-0xff). The effective depth range for this setup is then 0x010000 to 0xffffff.

4. Concentrating on the visible part

In the previous section we discussed how important the setting of near and far clipping is with respect to the depth resolution. For the shadow map resolution (width and height) the remaining four sides of the light's viewing frustum are crucial.

Consider a very large scene and a spotlight with a large cutoff angle. If we would just render the shadow map using the cutoff angle to determine the view frustum we would receive very coarse shadow edges when the camera focuses on small portions of the scene. Hence it is important that the viewing frustum of the light is optimized for the current camera view. This can be achieved by determining the visible pixels (as seen from the camera) and constructing a viewing frustum that includes all these relevant pixels.

In order to compute the visible pixels, we first render the scene from the camera position and use projective texturing to map a control texture onto the scene. This control texture is projected from the light source position and contains color-coded information about the row-column position, similar to the ramp texture used for the depth replace. In this step we use the maximal light frustum (cutoff angle) in order to ensure that all illuminated parts of the scene are processed. Since pixels outside the light frustum are not relevant we reserve one row-column entry, e.g. $(0, 0)$, for outside regions and use this as the texture's border color. By reading back the frame buffer to host memory we can now analyze which regions in the shadow map are used (exactly those for which we find row-column positions in the frame buffer). In the following subsections we will discuss methods for finding a suitable light frustum based on this information.

4.1. Axis aligned bounding rectangle

The easiest and fastest method is to compute the axis aligned bounding rectangle that encloses all relevant pixels. This can be implemented by searching for the maximum and minimum row and column values, while leaving out the values used for the outside part (texture border). This bounding rectangle can now be used to focus the shadow map on the visible pixels in the scene. All we have to do is to perform a scale and bias on the x and y coordinates after the light's projection matrix to bring

$$[x_{min}; x_{max}] \times [y_{min}; y_{max}] \rightarrow [-1; 1] \times [-1; 1] \qquad .$$

4.2. Optimal bounding rectangle

A better solution for adjusting the view of the light source is to compute the optimal bounding rectangle that encloses all visible pixels. This can be realized by using a method known as the *rotating calipers* algorithm [9, 6] which is capable of computing the minimum area enclosing rectangle in linear time. We start by computing the two dimensional convex hull of all visible points using the *monotone chain* algorithm proposed

by [1]. This algorithm is linear with respect to the number of input points $O(n)$, assuming that input points are sorted by increasing x and increasing y coordinates. As stated by [6], the minimum area rectangle enclosing a convex polygon P has a side collinear with an edge of P. Using this property, a brute-force approach would be to construct an enclosing rectangle for each edge of P. This has a complexity of $O(n^2)$ since we have to find minima and maxima for each edge separately. The rotating calipers algorithm rotates two sets of parallel lines (calipers) *around* the polygon and incremently updates the extreme values, thus requiring only linear time to find the optimal bounding rectangle. Figure 5 illustrates one step of this algorithm: The support lines are rotated (clockwise) until a line coincides with an edge of P. If the area of the new bounding rectangle is less than the stored minimum area rectangle, this bounding rectangle becomes the new minimum. This procedure is repeated until the accumulated rotation angle is greater than 90 degrees [2].

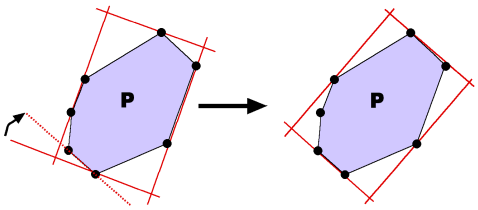

Figure 5. Rotating calipers.

5. Results

We have implemented the described optimizations using OpenGL and an NVIDIA GeForce4 card. For analyzing the scene, we first render from the camera position using only a small portion of the frame buffer (e.g. 64×64 pixels). During this step we apply the control texture described in Section 4 and store the resulting row and column information in the red and green channel. In order to determine the settings for the near and far plane (Section 3), we use an additional 2D ramp texture that encodes the distance to the light source in the blue and alpha channel. After this step, we read back the contents of the frame buffer (64×64 region) and process this control information in order to obtain the minimal enclosing rectangle (rotating calipers method) and the settings for the light source's near and far clipping plane.

Figure 6 shows an example scene illuminated by one spotlight with a large cutoff angle. Here, the image resolution was set to 512×512 pixels (4-times oversampling), whereas the shadow map only has a resolution of 256×256 pixels. For the control rendering pass a 64×64 pixel region was used. The frame rates for this scene are about 20 to 25 frames per second (74000 triangles).

In the left image in Figure 6 we directly compared the adjusted light frustum (left half of the image) with the result obtained using some fixed setting (right half). Here the adjustment can only slightly improve the shadow quality (coarse shadow edges), but

[2] For a detailed description of the algorithm please see [6].

the algorithm still computes an optimal setting for the light's near and far clipping plane. The top right image shows the optimized light frustum and the camera frustum, seen from a different perspective. In the bottom right image the convex hull and the resulting minimum area enclosing rectangle are drawn as they are located in the non-optimized shadow map.

In Figure 7 the camera was moved so that it focuses on a small part of the scene. Here the automatic adjustment greatly improves shadow quality since the shadow map now also focuses on the important part of the scene. It can be seen that the light frustum is slightly over-estimated. This due to the resolution of the control texture. An even tighter fit can be achieved by repeating the control texture rendering several times, so that the frustum converges near the optimum.

6. Conclusion

In this paper we have shown how the accuracy of shadow mapping can be greatly enhanced using a light source frustum that adapts to the actual camera view and depth values that are uniformly spaced between the light source's near and far plane. Since we clamp depth values rather than clipping geometry we can include shadow casters that are not visible from the camera view without stretching the light frustum, thus restricting depth precision to the visible part.

References

[1] A. Andrew. Another efficient algorithm for convex hulls in two dimensions. In *Info. Proc. Letters 9*, 1997.

[2] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg. Adaptive shadow maps. In *Proceedings of ACM SIGGRAPH 2001*, August 2001.

[3] W. Heidrich. *High-quality Shading and Lighting for Hardware-accelerated Rendering*. PhD thesis, University of Erlangen, Computer Graphics Group, 1999.

[4] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. *Proceedings of SIGGRAPH 2001*, August 2001.

[5] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, 2002. Available from http://www.nvidia.com.

[6] H. Pirzadeh. Computational geometry with the rotating calipers. Master's thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, November 1999.

[7] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadow and lighting effects using texture mapping. In *Proceedings of SIGGRAPH 1992*, July 1992.

[8] M. Stamminger and G. Drettakis. Perspective shadow maps, 2002. To appear in Proceedings of ACM SIGGRAPH 2002.

[9] G. T. Toussaint. Solving geometric problems with the rotating calipers. In *Proceedings of IEEE MELECON'83, Athens, Greece*, May 1983.

[10] L. Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, Aug. 1978.

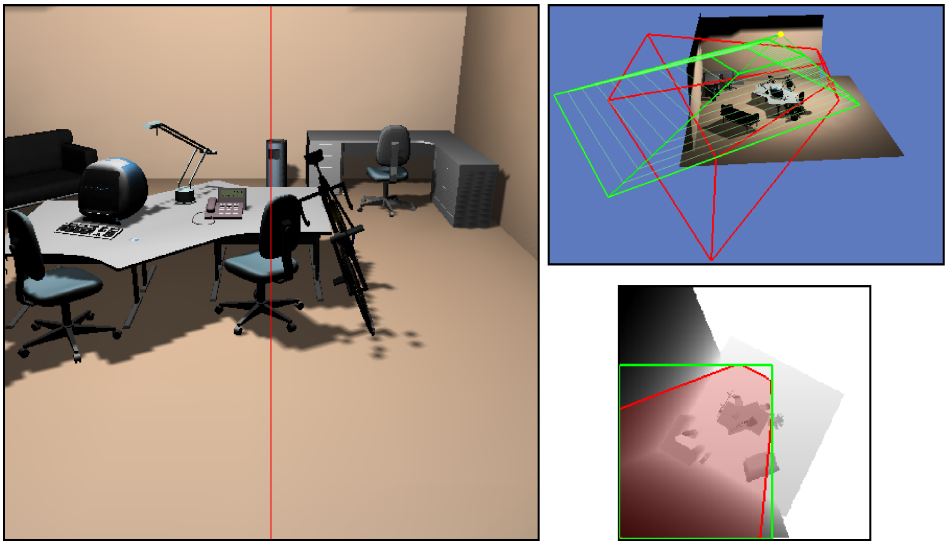

Figure 6. Left: Scene rendered with light frustum adjusted (left half) compared to the traditional method (right half). Top right: The optimized light frustum (green) and camera frustum (red). Bottom right: Location of convex hull (red) and minimum area enclosing rectangle (green) in the non-optimized shadow map.

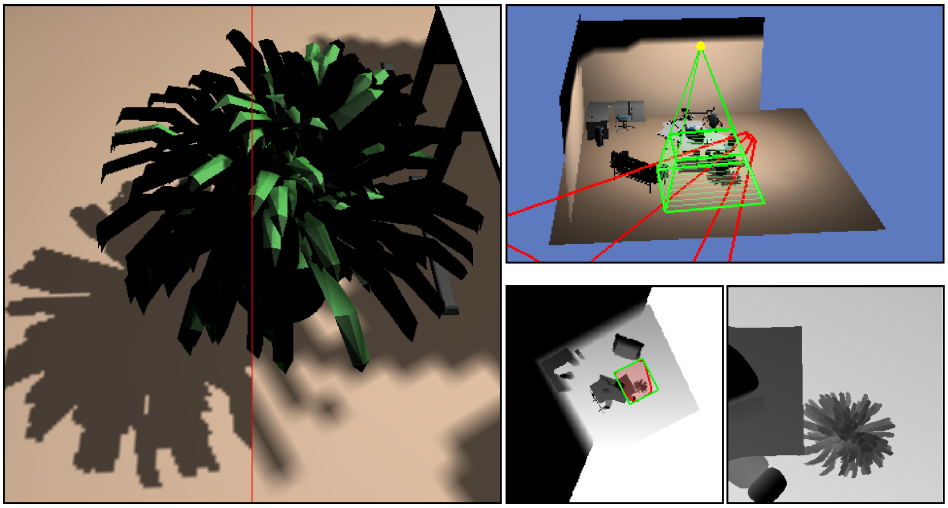

Figure 7. Left: Camera close-up (left half with adjusted light frustum, right half without). Top right: Optimized light frustum (green) and camera frustum (red). Bottom right: non-optimized shadow map and optimized shadow map.